**NAME:- SHAURYA PRAKASH SHAH**

**ROLL:- 001811001025**

**ASSIGNMENT – 5**

**IT 4TH YEAR 1ST SEM**

IPYNB Notebook Link:- https://colab.research.google.com/drive/13jQE77EuFZxJsO72X59Od4jMGGmmBskS?usp=sharing

GITHUB Link:- https://github.com/shauryashah/ML-Lab-Assignments.git

1. MOUNTAIN CAR - REINFORCEMENT LEARNING

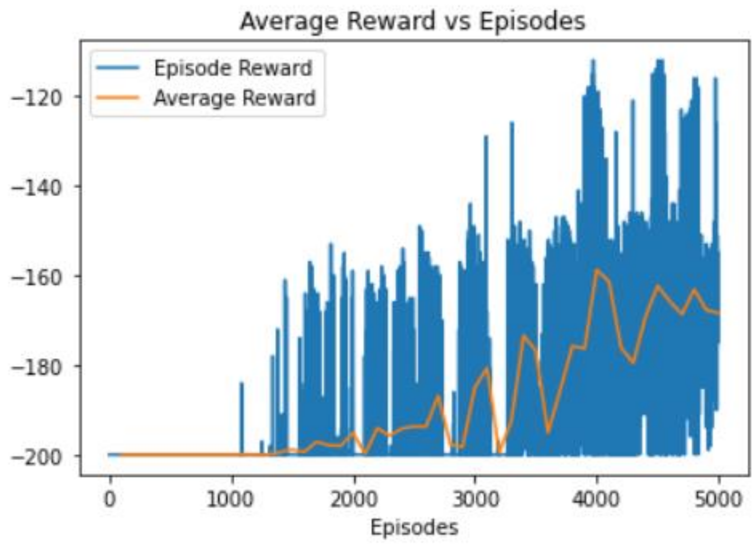
CODE

```
import gym
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Reshape, Conv2D, Dense, Flatten, BatchNormalization, Dropout, MaxPooling2D
from tensorflow.keras.optimizers import Adam, SGD

from rl.agents import DQNAgent
from rl.policy import EpsGreedyQPolicy
from rl.memory import SequentialMemory


def plot_average_reward(reward_list, ave_reward_list):
  plt.plot(np.arange(len(reward_list)), reward_list, label='Episode Reward')
  plt.plot(100*(np.arange(len(ave_reward_list)) + 1), ave_reward_list, label='Average Reward')
  plt.legend(loc='upper left')
  plt.xlabel('Episodes')
  plt.title('Average Reward vs Episodes')
  plt.show()
```

```
env = gym.make('MountainCar-v0')
env.reset()


print('State space: ', env.observation_space)
print('Action space: ', env.action_space)
print(env.observation_space.low)
print(env.observation_space.high)


%%time

learning = 0.1
discount = 0.95
epsilon = 0.5
min_eps = 0.0
episodes = 5000
epsilon_decay = (epsilon-min_eps)/episodes
reward_list = []
ave_reward_list = []
win_count = 0
cumu_timesteps = 0

discrete_obs_size = [20]*len(env.observation_space.high)
discrete_obs_window = (env.observation_space.high-
env.observation_space.low)/discrete_obs_size
q_table = np.random.uniform(low=-
2, high=0, size=(discrete_obs_size+[env.action_space.n]))

def get_discrete_state(state):
  discrete_state = (state-
env.observation_space.low)/discrete_obs_window
  return tuple(discrete_state.astype(np.int))


for episode in range(episodes):
  discrete_state = get_discrete_state(env.reset())
  tot_reward = 0
  done=False
  while not done:

    cumu_timesteps+=1
    if np.random.random() > epsilon:
      action = np.argmax(q_table[discrete_state])
    else:
      action = np.random.randint(0, env.action_space.n)

    new_state, reward, done, _ = env.step(action)
    new_discrete_state = get_discrete_state(new_state)
```

```
        if not done:
          max_future_q = np.max(q_table[new_discrete_state])
          current_q = q_table[discrete_state + (action,)]
          new_q = (1-
learning)*current_q + learning*(reward + discount*max_future_q)
          q_table[discrete_state + (action,)] = new_q

        elif new_state[0] >= env.goal_position:
          win_count+=1
          q_table[discrete_state + (action,)] = 0

        tot_reward+=reward
        discrete_state = new_discrete_state

    if epsilon > min_eps:
      epsilon-=epsilon_decay
    reward_list.append(tot_reward)

    if (episode+1) % 100 == 0:
        ave_reward = np.mean(reward_list[episode-99:])
        ave_reward_list.append(ave_reward)

    if (episode+1) % 500 == 0:
        print('Episode {} Average Reward: {}'.format(episode+1, ave_rewar
d))
env.close()
```

```
Episode 500 Average Reward: -200.0
Episode 1000 Average Reward: -200.0
Episode 1500 Average Reward: -199.2
Episode 2000 Average Reward: -197.14
Episode 2500 Average Reward: -195.4
Episode 3000 Average Reward: -186.4
Episode 3500 Average Reward: -168.82
Episode 4000 Average Reward: -159.06
Episode 4500 Average Reward: -177.29
Episode 5000 Average Reward: -161.02
CPU times: user 1min 52s, sys: 4.76 s, total: 1min 57s
Wall time: 1min 51s
```




2. MOUNTAIN CAR – DEEP REINFORCEMENT LEARNING

CODE

```
#DEFINING ACTION AND STATE SIZE
nb_actions = 3
nb_states = 2

#DEFINE DEEP LEARNING MODEL
model = Sequential()
model.add(Flatten(input_shape=(1,) + env.observation_space.shape))
model.add(Dense(128, activation='relu'))
model.add(Dense(512, activation="relu"))
model.add(Dropout(0.5))
model.add(Dense(3, activation="relu"))
model.summary()

#INITIALISE AGENT WITH EPSILON GREDDY POLICY
policy = EpsGreedyQPolicy()
memory = SequentialMemory(limit=5000, window_length=1)
agent = DQNAgent(model=model, memory=memory, policy=policy, nb_actions=
nb_actions,
                 nb_steps_warmup=500, target_model_update=1e-2)
agent.compile(Adam(lr=1e-3), metrics=['mse'])


#AGENT IS TRAINED ON ENIRONMENT
agent.fit(env, nb_steps=50000, visualize=False, verbose=1, nb_max_episo
de_steps=1000)
```

```
#AGENT IS TESTED FOR 10 EPISODES
agent.test(env, nb_episodes=10, nb_max_episode_steps=1000, visualize=False)
```

**OUTPUT**

```
Model: "sequential_14"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 2)                 0

 dense_33 (Dense)            (None, 128)               384

 dense_34 (Dense)            (None, 512)               66048

 dropout_10 (Dropout)        (None, 512)               0

 dense_35 (Dense)            (None, 3)                 1539

=================================================================
Total params: 67,971
Trainable params: 67,971
Non-trainable params: 0
_________________________________________________________________
```

```
Training for 50000 steps ...
Interval 1 (0 steps performed)
/usr/local/lib/python3.7/dist-packages/keras/engine/training_v1.py:2079: UserWarning
  updates=self.state_updates,
10000/10000 [==============================] - 118s 12ms/step - reward: -1.0000
50 episodes - episode_reward: -200.000 [-200.000, -200.000] - loss: 0.500 - mse: 0.3

Interval 2 (10000 steps performed)
10000/10000 [==============================] - 122s 12ms/step - reward: -1.0000
50 episodes - episode_reward: -200.000 [-200.000, -200.000] - loss: 0.500 - mse: 0.3

Interval 3 (20000 steps performed)
10000/10000 [==============================] - 123s 12ms/step - reward: -1.0000
50 episodes - episode_reward: -200.000 [-200.000, -200.000] - loss: 0.500 - mse: 0.3

Interval 4 (30000 steps performed)
10000/10000 [==============================] - 120s 12ms/step - reward: -1.0000
50 episodes - episode_reward: -200.000 [-200.000, -200.000] - loss: 0.500 - mse: 0.3

Interval 5 (40000 steps performed)
10000/10000 [==============================] - 124s 12ms/step - reward: -1.0000
done, took 606.494 seconds
<keras.callbacks.History at 0x7f5a9be0be50>
```

```
Testing for 10 episodes ...
Episode 1: reward: -200.000, steps: 200
Episode 2: reward: -200.000, steps: 200
Episode 3: reward: -200.000, steps: 200
Episode 4: reward: -200.000, steps: 200
Episode 5: reward: -200.000, steps: 200
Episode 6: reward: -200.000, steps: 200
Episode 7: reward: -200.000, steps: 200
Episode 8: reward: -200.000, steps: 200
Episode 9: reward: -200.000, steps: 200
Episode 10: reward: -200.000, steps: 200
<keras.callbacks.History at 0x7f5a9bafca90>
```

## 3. ROULETTE – REINFORCEMENT LEARNING

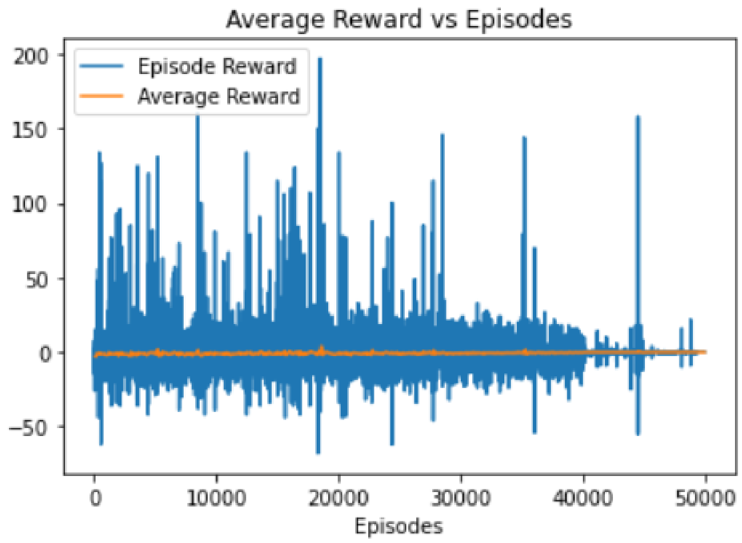
CODE

```
env = gym.make('Roulette-v0')
env.reset()
print('State space: ', env.observation_space)
print('Action space: ', env.action_space)


learning = 0.1
discount = 0.95
epsilon = 0.5
min_eps = 0.0
episodes = 50000
epsilon_decay = (epsilon-min_eps)/episodes
reward_list = []
ave_reward_list = []

q_table = np.random.randn(env.observation_space.n, env.action_space.n)
for episode in range(episodes):
  state = env.reset()
  tot_reward = 0
  done=False
  while not done:
    if np.random.random() > epsilon:
      action = np.argmax(q_table[state,:])
    else:
      action = np.random.randint(0, env.action_space.n)

    new_state, reward, done, _ = env.step(action)
    max_future_q = np.max(q_table[new_state, :])
    current_q = q_table[state, action]
    new_q = (1-
learning)*current_q + learning*(reward + discount*max_future_q)
    q_table[state, action] = new_q
```

```
    tot_reward+=reward
    state = new_state

  if epsilon > min_eps:
    epsilon-=epsilon_decay
  reward_list.append(tot_reward)

  if (episode+1) % 100 == 0:
      ave_reward = np.mean(reward_list[episode-99:])
      ave_reward_list.append(ave_reward)
      #reward_list = []

  if (episode+1) % 500 == 0:
      print('Episode {} Average Reward: {}'.format(episode+1, ave_rewar
d))
env.close()
```

## OUTPUT

```
Episode 35500 Average Reward: -1.22
Episode 36000 Average Reward: -0.37
Episode 36500 Average Reward: -0.28
Episode 37000 Average Reward: -0.76
Episode 37500 Average Reward: -0.23
Episode 38000 Average Reward: -0.32
Episode 38500 Average Reward: -0.53
Episode 39000 Average Reward: -0.11
Episode 39500 Average Reward: -0.11
Episode 40000 Average Reward: -0.17
Episode 40500 Average Reward: -0.07
Episode 41000 Average Reward: -0.03
Episode 41500 Average Reward: -0.12
Episode 42000 Average Reward: 0.0
Episode 42500 Average Reward: -0.05
Episode 43000 Average Reward: -0.04
Episode 43500 Average Reward: 0.0
Episode 44000 Average Reward: 0.05
Episode 44500 Average Reward: -0.05
Episode 45000 Average Reward: -0.26
Episode 45500 Average Reward: -0.06
Episode 46000 Average Reward: -0.01
Episode 46500 Average Reward: -0.01
Episode 47000 Average Reward: -0.02
Episode 47500 Average Reward: -0.01
Episode 48000 Average Reward: 0.01
Episode 48500 Average Reward: -0.01
Episode 49000 Average Reward: 0.0
Episode 49500 Average Reward: 0.0
Episode 50000 Average Reward: 0.0
CPU times: user 42.8 s, sys: 4.92 s, total: 47.7 s
Wall time: 43 s
```




## 4. ROULETTE – DEEP REINFORCEMENT LEARNING

### CODE

```
env = gym.make('Roulette-v0')
print(env.observation_space)
print(env.observation_space.shape)
print(env.action_space)
print(env.action_space.shape)


#DEFINE ACTION AND STATES
nb_actions = 38
nb_states = 1

#DEFINE DEEP LEARNING MODEL
model = Sequential()
model.add(Dense(128, input_dim=nb_states))
model.add(Dense(256, activation="relu"))
model.add(Dropout(0.5))
model.add(Dense(38, activation="relu"))
model.summary()

#DEFINE AGENT WITH EPSILON GREEDY POLICY
policy = EpsGreedyQPolicy()
memory = SequentialMemory(limit=5000, window_length=1)
agent = DQNAgent(model=model, memory=memory, policy=policy, nb_actions=
nb_actions,
                 nb_steps_warmup=500, target_model_update=1e-2)
agent.compile(Adam(lr=1e-3), metrics=['mse'])
```

```
#TRAIN AGENT ON ENVIRONMENT
agent.fit(env, nb_steps=50000, visualize=False, verbose=1, nb_max_episo
de_steps=1000)

#TEST AGENT ON 10 EPISODES
agent.test(env, nb_episodes=10, nb_max_episode_steps=1000, visualize=Fa
lse)
```

## OUTPUT

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_8 (Dense)             (None, 128)               256

 dense_9 (Dense)             (None, 256)               33024

 dropout_2 (Dropout)         (None, 256)               0

 dense_10 (Dense)            (None, 38)                9766

=================================================================
Total params: 43,046
Trainable params: 43,046
Non-trainable params: 0
_________________________________________________________________
```

```
Training for 50000 steps ...
Interval 1 (0 steps performed)
   53/10000 [..............................] - ETA: 9s - reward: -0.1509  /usr/local
  updates=self.state_updates,
10000/10000 [==============================] - 97s 10ms/step - reward: 0.0032
109 episodes - episode_reward: 0.394 [-98.000, 245.000] - loss: 17.272 - mse: 0.909

Interval 2 (10000 steps performed)
10000/10000 [==============================] - 104s 10ms/step - reward: 0.0165
119 episodes - episode_reward: 1.487 [-96.000, 132.000] - loss: 16.026 - mse: 0.843

Interval 3 (20000 steps performed)
10000/10000 [==============================] - 103s 10ms/step - reward: 0.0272
114 episodes - episode_reward: 2.254 [-96.000, 136.000] - loss: 16.692 - mse: 0.879

Interval 4 (30000 steps performed)
10000/10000 [==============================] - 101s 10ms/step - reward: -0.0149
114 episodes - episode_reward: -1.386 [-92.000, 245.000] - loss: 16.559 - mse: 0.872

Interval 5 (40000 steps performed)
10000/10000 [==============================] - 101s 10ms/step - reward: 0.0540
done, took 506.263 seconds
<keras.callbacks.History at 0x7f5a9cbaed90>
```

```
Testing for 10 episodes ...
Episode 1: reward: -26.000, steps: 100
Episode 2: reward: 11.000, steps: 100
Episode 3: reward: -63.000, steps: 100
Episode 4: reward: -26.000, steps: 100
Episode 5: reward: 11.000, steps: 100
Episode 6: reward: 11.000, steps: 100
Episode 7: reward: -26.000, steps: 100
Episode 8: reward: -63.000, steps: 100
Episode 9: reward: -26.000, steps: 100
Episode 10: reward: 85.000, steps: 100
<keras.callbacks.History at 0x7f5a9b877f10>
```

## 5. CAR RACING – REINFORCEMENT LEARNING

Car Racing was not implemented using q learning as the q table would be very large of the order of action space length * 256^(96*96*3).
Since action space is not discrete but continuous, hence the dimensions would be greater(equal to no of buckets required. **Hence, the RAM on Colab kept crashing while trying to train.**
Also each state of the car racing game is a snapshot of the current status of the game. Normal q learning methods are not good enough to train on this. A CNN would have much better results.

## 6. CAR RACING – DEEP REINFORCEMENT LEARNING

CODE

```
env = gym.make('CarRacing-v0')

print(env.observation_space)

print(env.observation_space.shape)

print(env.action_space)

print(env.action_space.shape)
```

```python
#DEFINE WRAPPER CLASS FOR CARRACING-V0 TO MAKE ACTIONS DISCRETE
class CarRacingDiscrit:

    def __init__(self):
        self.env = gym.make('CarRacing-v0')
        self.action_space = 10*10*10
        self.observation_space = 96*96*3

    def step(self, action):
        v1 = int(      action         ) % 10
        v2 = int( int(action) / 10  ) % 10
        v3 = int( int(action) / 100 ) % 10
        v1 = ( v1 - 5 ) / 5
        v2 = ( v2     ) / 10
        v3 = ( v3     ) / 10
        state, reward, done, info = self.env.step([v1, v2, v3])
        return state, reward, done, info

    def seed(self, s):
        return env.seed(s)

    def reset(self):
        return self.env.reset()
    def render(self):
        return self.env.render()

    def close(self):
        return self.env.close()
```

```python
env = CarRacingDiscrit()
nb_actions = 10*10*10
print(env.observation_space)
print(env.action_space)

#DEFINE DEEP LEARNING MODEL
model = Sequential()
model.add(Reshape((96, 96, 3), input_shape=(1, 96, 96, 3)))
model.add(BatchNormalization())
model.add(Conv2D(filters=32, kernel_size=(3, 3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(filters=64, kernel_size=(3, 3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(192, activation="relu"))
model.add(Dropout(0.5))
model.add(Dense(1000, activation="relu"))
model.summary()

#DEFINE AGENT WITH EPSILON GREEDY POLICY
policy = EpsGreedyQPolicy()
memory = SequentialMemory(limit=5000, window_length=1)
agent = DQNAgent(model=model, memory=memory, policy=policy, nb_actions=
nb_actions,
                 nb_steps_warmup=500, target_model_update=1e-2)
agent.compile(Adam(lr=1e-3), metrics=['mse'])

#TRAIN AGENT ON ENVIRONMENT
agent.fit(env, nb_steps=10000, visualize=False, verbose=1, nb_max_episo
de_steps=1000)
```

```
#TEST AGENT FOR 10 EPISODES
agent.test(env, nb_episodes=10, nb_max_episode_steps=1000, visualize=Fa
lse)
```

## OUTPUT

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 reshape (Reshape)           (None, 96, 96, 3)         0

 batch_normalization (BatchN  (None, 96, 96, 3)        12
 ormalization)

 conv2d (Conv2D)             (None, 94, 94, 32)        896

 max_pooling2d (MaxPooling2D  (None, 47, 47, 32)       0
 )

 batch_normalization_1 (Batc  (None, 47, 47, 32)       128
 hNormalization)

 conv2d_1 (Conv2D)           (None, 45, 45, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 22, 22, 64)       0
 2D)

 flatten (Flatten)           (None, 30976)             0

 dense (Dense)               (None, 192)               5947584

 dropout (Dropout)           (None, 192)               0

 dense_1 (Dense)             (None, 1000)              193000

=================================================================
Total params: 6,160,116
Trainable params: 6,160,046
Non-trainable params: 70
```

```
Training for 10000 steps ...
Track generation: 1163..1458 -> 295-tiles track
Interval 1 (0 steps performed)
/usr/local/lib/python3.7/dist-packages/keras/engine/training_v1.py:2079: UserWarnin
  updates=self.state_updates,
 1000/10000 [==>...........................] - ETA: 10:01 - reward: -0.0796Track ge
 2000/10000 [=====>........................] - ETA: 12:12 - reward: -0.0319Track ge
 3000/10000 [========>.....................] - ETA: 11:39 - reward: -0.0324Track ge
 4000/10000 [===========>..................] - ETA: 10:27 - reward: -0.0432Track ge
 5000/10000 [==============>...............] - ETA: 8:58 - reward: -0.0445Track gen
 6000/10000 [=================>............] - ETA: 7:18 - reward: -0.0433Track gen
 7000/10000 [====================>.........] - ETA: 5:32 - reward: -0.0421Track gen
 8000/10000 [=======================>......] - ETA: 3:43 - reward: -0.0421Track gen
 9000/10000 [==========================>...] - ETA: 1:52 - reward: -0.0447Track gen
10000/10000 [==============================] - 1157s 113ms/step - reward: -0.0419
done, took 1159.031 seconds
<keras.callbacks.History at 0x7f8f803a4490>
```

---

```
Testing for 10 episodes ...
Track generation: 1186..1490 -> 304-tiles track
Episode 1: reward: -83.498, steps: 1000
Track generation: 1177..1476 -> 299-tiles track
Episode 2: reward: -83.221, steps: 1000
Track generation: 1108..1389 -> 281-tiles track
Episode 3: reward: -82.143, steps: 1000
Track generation: 1004..1265 -> 261-tiles track
Episode 4: reward: -80.769, steps: 1000
Track generation: 1111..1393 -> 282-tiles track
Episode 5: reward: -82.206, steps: 1000
Track generation: 1120..1413 -> 293-tiles track
Episode 6: reward: -82.877, steps: 1000
Track generation: 1132..1419 -> 287-tiles track
Episode 7: reward: -82.517, steps: 1000
Track generation: 1184..1493 -> 309-tiles track
Episode 8: reward: -83.766, steps: 1000
Track generation: 1041..1310 -> 269-tiles track
Episode 9: reward: -81.343, steps: 1000
Track generation: 1053..1326 -> 273-tiles track
Episode 10: reward: -81.618, steps: 1000
<keras.callbacks.History at 0x7f8e93fb3ad0>
```

---

## 2. USER INPUT GRAPH

CODE

ASSUMING A GRAPH IN THE SHAPE OF A GRID WITH MOVEMENT ALLOWED IN ALL DIRECTIONS EXCEPT ALONG THE DIAGONALS

```
import random, math, time
import numpy as np
from keras.models import Sequential
from keras.layers import *
from tensorflow.keras.optimizers import *

import matplotlib
#matplotlib.use("Agg")
import matplotlib.pyplot as plt
from matplotlib.image import imread
from matplotlib import rc, animation
from IPython import display
from IPython.display import HTML
%matplotlib inline



#DEFINING AN ENVIRONMENT FOR A USER INPUT GRAPH
class Environment:

  def __init__(self, grid_size):
      self.grid_size = grid_size

      self.cat = imread('start.png')
      self.mouse = imread('dest.jpg')
```

```
        #self.confetti = imread('https://image.ibb.co/ganuAA/tom-and-
jerry.png')
        self.dim = 1.5

        self.rewards = []

    def _update_state(self, action):
        state = self.state
        # 0 = left
        # 1 = right
        # 2 = down
        # 3 = up

        fy, fx, py, px = state
        old_d = abs(fx - px) + abs(fy - py)

        if action == 0:
            if px > 0:
                px -= 1
        if action == 1:
            if px < self.grid_size-1:
                px += 1
        if action == 2:
            if py > 0:
                py-= 1
        if action == 3:
            if py < self.grid_size-1:
                py += 1

        new_d = abs(fx - px) + abs(fy - py)
        self.d = old_d-new_d
        self.time = self.time - 1
```

```python
    return np.array([fy, fx, py, px])

def _get_reward(self):
  fruit_y, fruit_x, player_y, player_x = self.state
  if fruit_x == player_x and fruit_y == player_y: return 1
  if self.d == 1: return 1
  if self.d == 0: return -1
  if self.d == -1: return -1

def _is_over(self):
  fruit_y, fruit_x, player_y, player_x = self.state
  if self.time == 0: return True
  if fruit_x == player_x and fruit_y == player_y: return True
  return False

def step(self, action):
  self.state = self._update_state(action)
  reward = self._get_reward()
  self.rewards.append(reward)
  game_over = self._is_over()
  return self.state, reward, game_over

def render(self):
  # Note: there's no promises of efficieny with this method
  # If things are slow, remove it

  im_size = (self.grid_size,)*2
  state = self.state

  self.fig = plt.figure(figsize=(8, 6), dpi=80)
  self.ax = self.fig.add_subplot(111)
```

```
    self.ax.clear()
    self.ax.set_ylim((-1, self.grid_size))
    self.ax.set_xlim((-1, self.grid_size))
    #self.ax.axis('off') # uncomment to turn off axes
    self.ax.get_xaxis().set_ticks(range(self.grid_size))
    self.ax.get_yaxis().set_ticks(range(self.grid_size))

    xc = state[2]
    yc = state[3]
    xm = state[0]
    ym = state[1]

    if state[0] == state[2] and state[1] == state[3]:
      self.ax.imshow(self.cat,
                     extent=(-1, self.grid_size,
                             -1, self.grid_size))
    else:
      self.ax.imshow(self.mouse,
                     extent=(xm-self.dim/4, xm+self.dim/4,
                             ym-self.dim/4, ym+self.dim/4))
      self.ax.imshow(self.cat,
                     extent=(xc-self.dim/4, xc+self.dim/4,
                             yc-self.dim/4, yc+self.dim/4))
    self.fig.canvas.draw()
    return np.array(self.fig.canvas.renderer._renderer)

  def reset(self, deterministic=True):
    if deterministic:
      # this is an easier environment setup
      fruit_x = 0
```

```
        fruit_y = 0
        player_x = self.grid_size - 1
        player_y = self.grid_size - 1
        time = self.grid_size*2
      else:
        generated = False
        while not generated\
        or abs(fruit_x - player_x) + abs(fruit_y - player_y) < self.grid_
size/2:
          fruit_x = np.random.randint(0, self.grid_size-1)
          fruit_y = np.random.randint(0, self.grid_size-1)
          player_x = np.random.randint(0, self.grid_size-1)
          player_y = np.random.randint(0, self.grid_size-1)
          time = abs(fruit_x - player_x) + abs(fruit_y - player_y)
          time *= 2
          generated = True

      self.time = time
      self.d = 0
      self.state = np.asarray([fruit_y, fruit_x, player_y, player_x])

      return self.state


"""
This runs the environment using random actions
"""


print('Setting up environment')
env = Environment(5)
num_episodes = 1 # number of games we want the agent to play
env.reset()
```

```python
frames = []
RENDER = True
print('Running random simulation')
for episode in range(num_episodes):
  print('Resetting environment')
  s = env.reset() # Initial state
  while True:
    a = np.random.choice(range(4)) # choose a random action
    s_, r, done = env.step(a) # apply random action

    if RENDER:
      fig = env.render()
      plt.imshow(fig)
      plt.show()
      frames.append(fig)

    if done:
      break
```

## REINFORCEMENT LEARNING

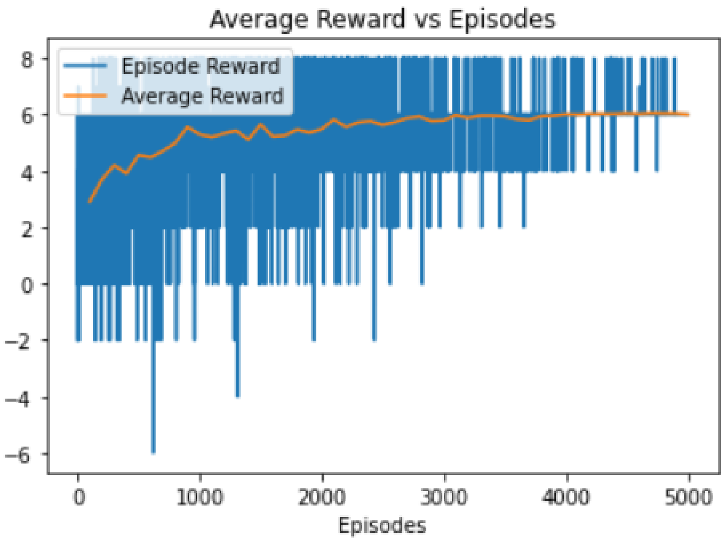
```python
%%time

learning = 0.1
discount = 0.95
epsilon = 0.5
min_eps = 0.0
episodes = 5000
epsilon_decay = (epsilon-min_eps)/episodes
reward_list = []
ave_reward_list = []
```

```
win_count = 0

cumu_timesteps = 0


discrete_obs_size = [5]*4

q_table = np.random.uniform(low=-
2, high=0, size=(discrete_obs_size+[4]))

print(q_table.shape)


def get_discrete_state(state):

  discrete_state = (state-
env.observation_space.low)/discrete_obs_window

  return tuple(discrete_state.astype(np.int))


for episode in range(episodes):

  state = tuple(env.reset().astype(np.int))

  tot_reward = 0

  done=False

  while not done:


    cumu_timesteps+=1

    if np.random.random() > epsilon:

      action = np.argmax(q_table[state])

    else:

      action = np.random.randint(0, 4)


    new_state, reward, done = env.step(action)

    new_state = tuple(new_state.astype(np.int))

    if not done:

      max_future_q = np.max(q_table[new_state])

      current_q = q_table[state + (action,)]

      new_q = (1-
learning)*current_q + learning*(reward + discount*max_future_q)
```

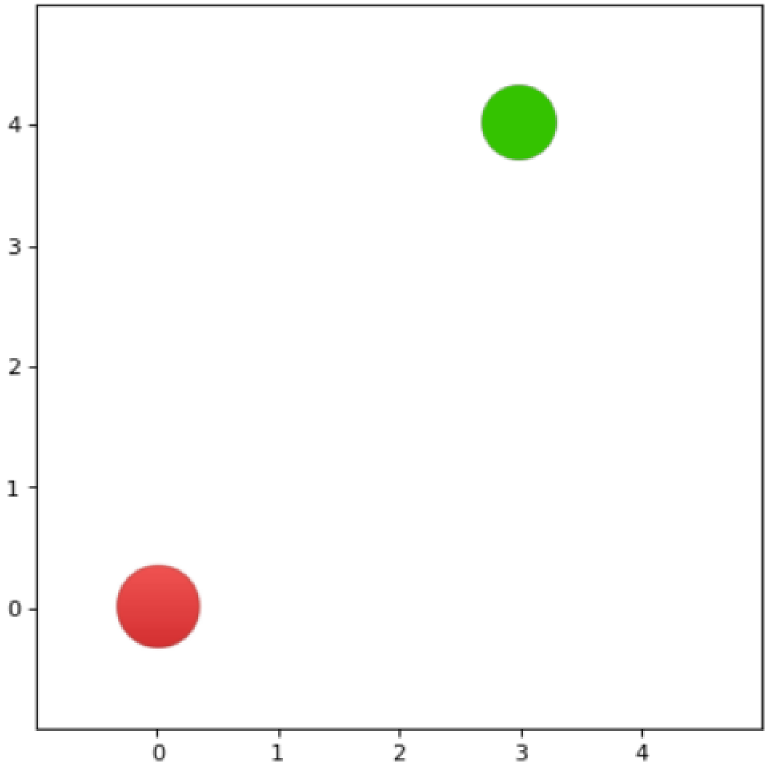
```
        q_table[state + (action,)] = new_q

      elif done:
        win_count+=1

      tot_reward+=reward
      state = new_state

    if epsilon > min_eps:
      epsilon-=epsilon_decay
    reward_list.append(tot_reward)

    if (episode+1) % 100 == 0:
        ave_reward = np.mean(reward_list[episode-99:])
        ave_reward_list.append(ave_reward)

    if (episode+1) % 500 == 0:
        print('Episode {} Average Reward: {}'.format(episode+1, ave_rewar
d))
print(win_count)
```

## OUTPUT

```
(5, 5, 5, 5, 4)
Episode 500 Average Reward: 4.56
Episode 1000 Average Reward: 5.3
Episode 1500 Average Reward: 5.65
Episode 2000 Average Reward: 5.48
Episode 2500 Average Reward: 5.63
Episode 3000 Average Reward: 5.79
Episode 3500 Average Reward: 5.94
Episode 4000 Average Reward: 6.01
Episode 4500 Average Reward: 6.04
Episode 5000 Average Reward: 6.0
5000
CPU times: user 1.68 s, sys: 117 ms, total: 1.8 s
Wall time: 1.69 s
```




```
Setting up environment
Running random simulation
Resetting environment
```




**DEEP REINFORCEMENT LEARNING**

**CODE**

```
#-------------------- BRAIN ---------------------------

class Brain:
  """The 'brain' of the agent, where the model is created and held.

  state_dim (int): the size of the observation space
  action_dim (int): the size of the action space

  """
  def __init__(self, state_dim, action_dim, weights=None):
    self.state_dim = state_dim
    self.action_dim = action_dim

    self.model = self._createModel()
    if weights:
      self.model.load_weights("brain.h5")

  def _createModel(self):
    # Creates a Sequential Keras model
    # This acts as the Deep Q-Network (DQN)

    model = Sequential()

    ### START CODE HERE ### (≈ 3 lines of code)
    # 'Dense' is the basic form of a neural network layer
    # Input Layer with activation function relu and Hidden Layer with 1
28 nodes
```

```
    model.add(Dense(128, input_dim=self.state_dim, activation='relu'))
    #Second Hidden layer with 128 nodes
    model.add(Dense(128, activation='relu'))
    #Output layer with activation linear.
    #action_size=4
    model.add(Dense(self.action_dim, activation='linear'))


    ### END CODE HERE ###

    opt = RMSprop(lr=0.00025)
    model.compile(loss='mse', optimizer=opt)

    return model

  def train(self, x, y, epoch=1, verbose=0):
    self.model.fit(x, y, batch_size=64, epochs=epoch, verbose=verbose)

  def predict(self, s):
    return self.model.predict(s)

  def predictOne(self, s):
    return self.predict(s.reshape(1, self.state_dim)).flatten()


#-------------------- MEMORY --------------------------
class Memory:   # stored as ( s, a, r, s_ )
  """The agent's 'memory', where experiences are stored
  """

  def __init__(self, capacity):
```

```
    self.capacity = capacity
    self.samples = []

  def add(self, sample):
    # a sample should be an array [s, a, r, s_]
    # s: current state
    # a: current action
    # r: current reward
    # s_: next state
    self.samples.append(sample)

    if len(self.samples) > self.capacity:
        self.samples.pop(0)

  def sample(self, n):
    n = min(n, len(self.samples))
    return random.sample(self.samples, n)


#-------------------- AGENT ---------------------------
import math
class Agent:
  """The agent, which learns to navigate the environment

  """

  def __init__(self, state_dim, action_dim, memory_capacity = 10000,
              batch_size = 64, gamma = 0.99, lamb = 0.001,
               max_epsilon = 1., min_epsilon = 0.01):
    self.state_dim = state_dim
    self.action_dim = action_dim
```

```
    self.batch_size = batch_size
    self.gamma = gamma # discount rate, to calculate the future discoun
ted reward
    self.lamb = lamb
    self.max_epsilon = max_epsilon
    self.epsilon = max_epsilon
    self.min_epsilon = min_epsilon

    self.brain = Brain(state_dim, action_dim)
    self.memory = Memory(memory_capacity)
    self.steps = 0
    self.epsilons = []

  def act(self, s, verbose=False):
    """The policy of the agent:
    Here, we determine if we explore (take a random action) based on ep
silon.
    If not, we have the model predict the Q-Values for the state,
    then take the action which maximizes those values.
    """
    if random.random() < self.epsilon:
      if verbose:
        print("Random Action.")
      return random.randint(0, self.action_dim-1)
    else:
      actions = self.brain.predictOne(s)
      if verbose:
        print("Actions:", actions)
      return np.argmax(actions)
```

```
  def observe(self, sample):  # in (s, a, r, s_) format
    """The agent observes an event.

    We pass a sample (state, action, reward, next state) to be stored i
n memory.

    We then increment the step count and adjust epsilon accordingly.

    """

    self.memory.add(sample)


    # slowly decrease Epsilon based on our eperience

    self.steps += 1


    ### START CODE HERE ### (≈ 1 line of code)


    self.epsilon=self.min_epsilon+(self.max_epsilon-
self.min_epsilon)* math.exp((-self.lamb)*abs(self.steps))

    #ϵ=ϵmin+(ϵmax−ϵmin)∗e−λ|S|


    ### END CODE HERE ###


    self.epsilons.append(self.epsilon)


  def replay(self):

    """The agent learns based on previous experiences.

    We sample observations (state, action, reward, next state) from mem
ory.

    We train the model based on these observations.

    """


    # Random sample of experiences

    batch = self.memory.sample(self.batch_size)

    batch_size = len(batch)
```

```
    # Extracting states ('current' and 'next') from samples
    no_state = np.zeros(self.state_dim)
    states = np.array([ o[0] for o in batch ])
    states_next = np.array([ (no_state if o[3] is None else o[3]) for o
in batch ])



    # Estimating Q-Values for states
    q_vals = self.brain.predict(states)
    q_vals_next = self.brain.predict(states_next)



    # Setting up training data
    x = np.zeros((batch_size, self.state_dim))
    y = np.zeros((batch_size, self.action_dim))
    done=False
    for i in range(batch_size):
        obs = batch[i]
        st = obs[0];
        act = obs[1];
        rew = obs[2];
        st_next = obs[3]
        t = q_vals[i]

      ### START CODE HERE ### (≈ 4 line of code)
        if st_next is None:
            t[act]=rew
        else:
            t[act] = (rew + self.gamma *np.amax(q_vals_next[i]))
```

```
        ### END CODE HERE ###

        # Set training data
          x[i] = st
          y[i] = t

      # Train
      self.brain.train(x, y)



#-------------------- MAIN ----------------------------
print('Setting up environment')
env = Environment(5)

state_dim = 4
action_dim = 4 # left, right, up, down
print('Setting up agent')
MAX_EPSILON = 1 # the rate in which an agent randomly decides its actio
n
MIN_EPSILON = 0.05 # min rate in which an agent randomly decides its ac
tion
LAMBDA = 0.00005      # speed of decay for epsilon
num_episodes = 10000 # number of games we want the agent to play

VERBOSE = False
agent = Agent(state_dim, action_dim, lamb=LAMBDA,
              max_epsilon=MAX_EPSILON, min_epsilon=MIN_EPSILON)
env.reset()
episode_rewards = []
epsilons = []
t0 = time.time()
frames = []
```

```python
print('Running simulation')
for episode in range(num_episodes):
  s = env.reset() # Initial state
  if episode % 1000 == 0:
      fig = env.render()
      frames.append(fig)
  R = 0
  while True:
    a = agent.act(s, verbose=VERBOSE)

    s_, r, done = env.step(a)

    if done: # terminal state
        s_ = None

    agent.observe( (s, a, r, s_) )
    agent.replay()

    s = s_
    R += r

    if episode % 1000 == 0:
      fig = env.render()
      frames.append(fig)

    if VERBOSE:
      print("Action:", a)
      print("Reward:", r)

    if done:
```

```
    break

  epsilons.append(agent.epsilon)
  episode_rewards.append(R)

  if episode % 100 == 0:
    print('Episode {}'.format(episode))
    print('Time Elapsed: {0:.2f}s'.format(time.time() - t0))
    print('Epsilon {}'.format(epsilons[-1]))
    print('Last Episode Reward: {}'.format(R))
    print('Episode Reward Rolling Mean: {}'.format(np.mean(episode_rewa
rds[:-100])))
    print('-'*10)


agent.brain.model.save("brain.h5")
```

## OUTPUT

```
Setting up environment
Setting up agent
Running simulation
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/rmsprop.py:130: User
  super(RMSprop, self).__init__(name, **kwargs)
/usr/local/lib/python3.7/dist-packages/keras/engine/training_v1.py:2079: UserW
  updates=self.state_updates,
Episode 0
Time Elapsed: 1.91s
Epsilon 0.9995251187302109
Last Episode Reward: 0
Episode Reward Rolling Mean: nan
----------
/usr/local/lib/python3.7/dist-packages/numpy/core/fromnumeric.py:3373: Runtime
  out=out, **kwargs)
/usr/local/lib/python3.7/dist-packages/numpy/core/_methods.py:170: RuntimeWarn
  ret = ret.dtype.type(ret / rcount)
Episode 100
Time Elapsed: 12.45s
Epsilon 0.9532162322387105
Last Episode Reward: 6
Episode Reward Rolling Mean: 0.0
----------
Episode 200
Time Elapsed: 23.07s
Epsilon 0.9091658567921319
Last Episode Reward: 0
Episode Reward Rolling Mean: 0.2376237623762376
----------
Episode 300
Time Elapsed: 33.76s
Epsilon 0.8673455739778486
Last Episode Reward: 0
Episode Reward Rolling Mean: 0.527363184079602
----------
```

```
Episode 9400
Time Elapsed: 1025.64s
Epsilon 0.06291658293417288
Last Episode Reward: 7
Episode Reward Rolling Mean: 6.281367594882271
----------
Episode 9500
Time Elapsed: 1036.22s
Epsilon 0.06237108611129126
Last Episode Reward: 7
Episode Reward Rolling Mean: 6.295394107009892
----------
Episode 9600
Time Elapsed: 1046.45s
Epsilon 0.06185218198747368
Last Episode Reward: 8
Episode Reward Rolling Mean: 6.307546574044838
----------
Episode 9700
Time Elapsed: 1056.87s
Epsilon 0.061358450275701804
Last Episode Reward: 8
Episode Reward Rolling Mean: 6.320487449224039
----------
Episode 9800
Time Elapsed: 1067.66s
Epsilon 0.06087766913536543
Last Episode Reward: 8
Episode Reward Rolling Mean: 6.335429337181734
----------
Episode 9900
Time Elapsed: 1078.60s
Epsilon 0.060408387636363046
Last Episode Reward: 8
Episode Reward Rolling Mean: 6.3457810427507395
```

**COMPARISON**

MOUNTAIN CAR

| | Time taken | Episode count | Average Reward |
|---|---|---|---|
| Reinforcement Learning | 111 seconds | 5000 | -161.02 |
| Deep Reinforcement Learning | 606.494 seconds | 250 | -200 |

ROULETTE

| | Time taken | Episode count | Average Reward |
|---|---|---|---|
| Reinforcement Learning | 43 seconds | 50000 | 0 |
| Deep Reinforcement Learning | 568.864 seconds | 558 | -11.2 |

CAR RACING

| | Time taken | Step count | Average Reward |
|---|---|---|---|
| Deep Reinforcement Learning | 1159.031 seconds | 10000 | -81.618 |

USER INPUT GRAPH

| | Time taken | Step count | Average Reward |
|---|---|---|---|
| Reinforcement Learning | 1.69 seconds | 5000 | 6 |
| Deep Reinforcement Learning | 1078.60 seconds | 10000 | 6.345781027 |